

MAKE

PLANET BOMBERS

YOUR

COMING SOON

A GAME

DIVE BOMBER



PHILIP O'CARROLL

[illegible][illegible]

DIVE BOMBER

First published 1984

Pitman Publishing Pty Ltd
(Incorporated in Victoria)

158 Bouverie Street
Carlton
Victoria 3053

Level 12

Town Hall House
452-462 Kent Street

Sydney

New South Wales 2000

9th Floor

National Bank Building
420 George Street
Brisbane
Queensland 4000

© Philip O'Carroll 1984

National Library of Australia
Cataloguing in Publication Data

O'Carroll, Philip, 1945-

Make programming your Commodore 64 a game.

ISBN 0 85896 159 8.

1. Electronic games. 2. Commodore 64 (Computer) -
Programming. I. Title.

794.8'2

Produced for Pitman
by Lucas Comber McGrath Pty Ltd

Text set in 10 point Times
by H & M Typesetting & Graphics Pty Ltd
Melbourne

Printed in Singapore
by GnP Consultants Pte Ltd

Associated companies

Pitman Publishing Ltd
London

Copp Clark Pitman
Toronto

Pitman Learning Inc
Belmont, California

Pitman Publishing New Zealand Ltd
Wellington

Contents

Before you start	1
1 Introduction	3
2 Setting the scene	5
3 Creating the aircraft	10
4 Bombs away!	14
5 Pilot control	18
6 Landing	20
7 Improving the graphics	23
8 Scoring the game	26
9 Zaps and booms	30
10 Joystick control	35
11 Dive Bomber — the complete program	36
12 How to write your own computer program	39

Before you start

This is a book for beginners but it will take you much further, much faster, than most other books for beginners. It focuses on the development of just one game and it explains in detail every step needed for the creation of its graphics, its sound and its action.

Some games books give you listings for lots of programs and explain them in language only the experienced programmer can understand. With this book, you'll really learn how to program your Commodore 64. You'll be building up an exciting game and at the same time you'll be learning the real skills of programming. By the time you've worked your way through it, you'll understand enough of programming in BASIC on the Commodore 64 to be able to write your own games.

There are, of course, a few things you ought to know and a few things you ought to be able to do before you start. You should know how to switch on your computer, connect it to your cassette or disk drive and tune in your monitor. You should know how to SAVE a program and you should know the functions of the various keys on your keyboard. In particular, you should know how to correct typing mistakes using the cursor keys. Beyond that, this really is a book for beginners!

Before you start

Many of the things that beginners want to know soon after they've started are contained in the *Commodore 64 User's Guide*. If you've worked your way through chapters 1, 2 and 3 of the *User's Guide*, that will be quite enough for you to start on *Dive Bomber*. You should, of course, keep your *User's Guide* by your side in case you want to go more deeply into any of the programming techniques we use in *Dive Bomber*. It will continue to be a useful reference.

IMPORTANT IMPORTANT IMPORTANT IMPORTANT

Throughout the book you'll find many lines of computer code. When you type them in, your Commodore 64 will automatically start a new line after the 40th character, even if it's in the middle of a word. You should keep typing till you come to the end of the *printed* line before you press the RETURN key. Only press the RETURN key when you want to type a new line number. Aside from that, key every line *exactly* as it's shown. Even a comma left out could cause your program to crash. If you misspell a command or omit a key symbol, your computer will tell you SYNTAX ERROR when you try to RUN the program. But don't despair. It happens to experienced programmers as well as beginners. Just go back to the offending line, correct it and press the RETURN key at the end of the line. Then all will be well.

From time to time, in the lines of computer code, you'll find a variety of graphic symbols. Here's what they mean and how you get them:

"X"X"X"	CURSOR DOWN	VERTICAL CRSR
"P"X"X"	CURSOR RIGHT	HORIZONTAL CRSR
"X"X"X"	TOP OF SCREEN	CLR/HOME
"T"X"X"	CLEAR SCREEN	SHIFT & CLR/HOME
"X"X"X"	BLACK	CTRL & 1
"X"X"X"	RED	CTRL & 2
"X"X"X"	BLUE	CTRL & 7
"X"X"X"	BROWN	COMMODORE & 2

1

Introduction and program plan

Dive Bomber places you at the controls of a super jet. Your mission is to destroy the city that lies below you and in doing so, to make for yourself a strip long enough to land on. You can dive low to increase your accuracy as you release your deadly bombs but beware — you'll pay a heavy price if you don't pull out in time!

Dive Bomber includes a number of subroutines that are easily transportable to a variety of other action games. In setting up *Dive Bomber* you'll see how to:

- set up a background of city buildings;
- create an aircraft with full pilot controls enabling you to climb and dive, and land and taxi to a halt (when you've cleared a runway for it!);
- drop bombs from your aircraft;
- create realistic sound effects for bombs falling and exploding (and for your aircraft, should you crash it!);
- set up a system that keeps score for you;
- control the game with your joystick as well as your keyboard.

For each of the modules (subroutines) that we need, we'll start with a simple, 'bare bones' version and then, later on in the book, look at ways of making it smarter or more realistic looking.

Introduction and program plan

Further down this page, there's an outline of the structure of the whole program. In describing the step by step writing of the program, however, we don't always follow the same sequence that's given there. We skip around a bit. That's because the order of a finished program isn't always the best order to write it in. Certainly, it makes it easier if we plan the sorts of modules we're going to need — as we have in the list above — but there are going to be times when we won't know exactly what we'll need until we're well into our program. Fortunately, it's an easy matter then to write the extra lines and then slip them in, back where they were needed.

So read the part called *Structure of the program* to get a bird's eye view of all the modules we're going to need, then hop straight into Chapter 2 — *Setting the scene*.

Structure of the program — the program plan

This is a summary of the program for *Dive Bomber*. The complete listing of all the lines is given on pages 36 to 38.

Lines 50-56	set up the variable names for the sound effects and the background and border colors
Lines 90-95	color the screen and border, set the screen clear and print the name of the game top centre
Lines 100-117	display buildings
Lines 200-218	move the aircraft forward a step at a time and up or down as the pilot requires
Lines 220-242	detect collisions of the aircraft with buildings
Lines 250-272	taxi aircraft to land
Lines 300-320	create bomb movement and whine sound effect
Lines 350-362	check pilot action for bomb release
Line 400	returns the program to line 214, the aircraft flight sequence, after the check for bomb release has been made
Lines 600-710	create aircraft crash effect and boom sound
Lines 750-760	create bomb hit effect
Lines 800-844	print score, high score, and allow user to restart game

2

Setting the scene

2.1 Screen color and border

Our first step will be to set an appropriate color for the sky and to make the border a contrasting color. We'll use light blue for the sky and green for the border. Enter and RUN this line:

```
90 POKE53281,14:POKE53280,5
```

The number 53281 is the memory location for any changes you want to make to the screen color and the number 53280 is the location for color changes to the border. 14 is the color code for light blue and 5 is the color code for green. Your *Commodore 64 User's Guide* contains a complete list of the color codes so you can choose any combination you like.

Now press RUN/STOP and RESTORE (together) and type LIST so we can continue to add lines to our program. Our next line clears the screen and prints the name of the game, top centre of the screen.

```
95 PRINT"?", "***** DIVE BOMBER *"
```

The reverse heart symbol appears when you press the CLR/HOME key with SHIFT. This clears the screen. The other symbols that appear before the words DIVE BOMBER occur when you press:

- CTRL with BLUE
- down cursor (three times)
- right cursor (twice)
- star

Now RUN your two line program to check that it works. Then type LIST and carry on with the next section.

Setting the scene

2.2 Building buildings

To make our buildings, we'll need building blocks — blocks of solid color that will serve as the storeys of the buildings. There's no key labelled with a block but that won't stop us. Without giving it a line number, try this:

- hold down the CTRL key and press the RVS ON key (key number 9)
- press the space bar

What you get, of course, is a solid block of color. A solid block is the REVERSE of a space. (Press CTRL and RVS OFF to cancel this command.)

In the programs in this book, we're using the POKE method of displaying characters so we'll need to identify the space character by its character code number. The code for a space is 32. 32 won't be quite the right number for us, though, because what we want is a reverse space. The character number for any reverse character is the character number plus 128. So for our solid block, the number will be 160.

Now, of course, we've got to think about locating the blocks on the screen. The place for the buildings is the bottom of the screen. For the moment we'll use the bottom seven rows — ie row 24 up to row 18. Our buildings will all be the same height, but that doesn't matter: we'll change them later.

```
100 REM DISPLAY BUILDINGS
102 SC=1024:REM 1ST SCREEN ADDRESS
105 FOR CO = 0 TO 39
108 FOR RO = 18 TO 24
109 PL=SC+RO*40+CO
111 POKE PL,160
```

Line 100 doesn't actually do anything. It's just a 'header' for this module or subroutine. REM speaks only to the programmer.

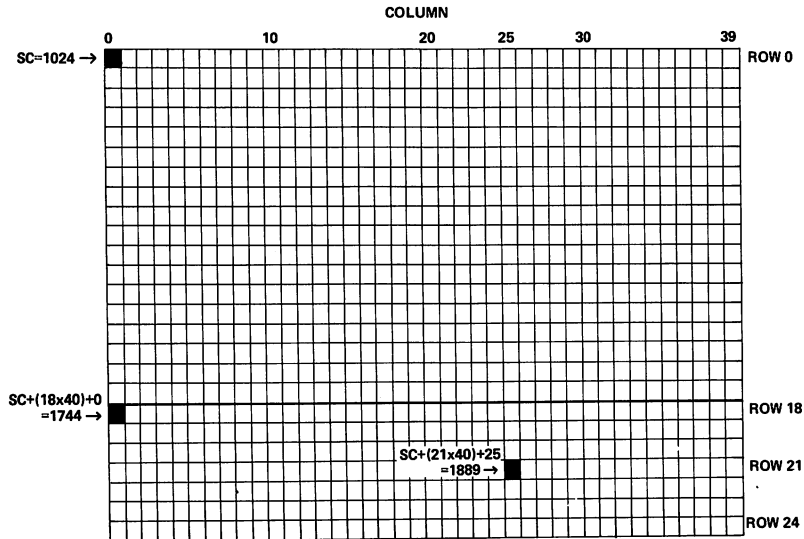
Line 102 sets up a variable name for the first screen address — ie we've called the first screen address (1024) by the name SScreen. So now when we want to locate a character on the screen, we don't have to look up the full map of screen addresses: we just have to call SC (short for screen) and add to it the number of places required to take us to the new screen address.

Line 105 sets up a variable name (ColumN) to name a position across the screen: 0 for the leftmost position and 39 for the rightmost. By writing FOR CO = 0 to 39, we are ensuring that our blocks will be POKEd into all 40 columns.

Setting the scene

Line 108 sets up a variable name (ROw) for the rows of the screen to be covered by the buildings. Our blocks will be POKEd into the rows from 18 down to 24.

Line 109 puts together all the data about the ROws and COlumnns where we want our buildings to appear and sets up a variable name for each position, PLace. The reasoning behind the formula for PLace is probably best seen in a diagram of the screen memory map.



So we see that the PLace our buildings will occupy, equals:

- SC (the first address on the screen) *plus*
- RO times 40 (where each of the row numbers from 18 down to 24 has been multiplied by 40 to give its actual screen address when it's added to 1024) *plus*
- CO the column position (across the screen) that we want filled with a building block.

Line 111 now POKes the character for reverse space (160) into each value of PL (short for PLace).

Setting the scene

2.3 Coloring the buildings

Lines 100 to 111 won't do anything for us until we add a line for the color of the blocks. The address for information about the *color* of a screen location occurs 54272 places further on than the address for the *character*. So now, if we want to color our buildings brown (color code 9), we add the line:

```
113 POKE PL+54272,9
```

And because our program contains two FORs (in lines 105 and 108) we'll have to finish with two NEXTs.

```
115 NEXT: NEXT
```

These complete the loops started in lines 105 and 108 — the loops that put Column through 40 different values and Row through the values 18 to 24. NEXT causes everything between the FOR and the NEXT to be repeated for the number of steps indicated in the line containing the word FOR.

Test as you go

It's a good idea to test run each of the modules as you complete them. It makes debugging a whole lot easier. So RUN lines 90 to 115 to check that they work as they should.

Variable names on the Commodore 64

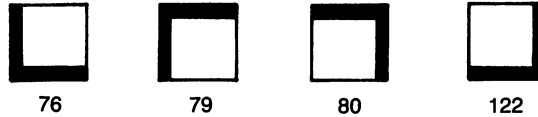
On the Commodore 64 you can use real names (like COLUMN, ROW and SCREEN) for the variables in your programs. The Commodore 64, however, only takes notice of the first two letters of the word. So if we use the word SCREEN, for example, we can't use the word SCORE later on in the program and expect the computer to distinguish it from SCREEN.

2.4 Separating the buildings

The blocks of brown that we've now got on our screen don't look much like buildings. That's because they're packed together too tightly. We really need slightly smaller blocks so the buildings will appear to stand apart from each other. There's no graphic key indicating a slightly reduced block but we can achieve the same effect by using one of the following characters and reversing

Setting the scene

it. In your mind's eye, imagine what each of these will look like when it's reversed:



I've chosen character 122 because it leaves a space below it and that will make the bottom storey appear to be separate from the ground. So for line 111, we substitute the following:

```
111 POKE PL,250
```

The character number is 250 because we need to add 128 to 122 to create the reverse. RUN the program again. Better? It's not perfect but it's enough for us to work with for the present. We're ready now to move on to displaying the aircraft.

3

Creating the aircraft

3.1 Enter the dive bomber

Our first step will be to choose a character to represent the aircraft. Just for the present we'll use the reverse of character number 124 (ie character number 252). It will look like this:



124 + 128

We'll start its flight path fairly high on the screen — say, row 4 — and in the first column. The first column is column 0. It's on the left hand side of the screen.

```
200 REM PLANE MOVEMENT
202 RO=4:CO=0
204 PL=SC+RO*40+CO
240 POKE PL,252
```

Notice that we've used the same variable names for ROW and COLUMN as we did earlier but that their values have changed. Consequently, PLACE is now also different.

And again, before we can make the aircraft appear, we'll have to give it a color. We'll make it black so it will stand out sharply. The color code for black is 0. The color address is 54272 places on.

```
242 POKE PL+54272,0
```

Now test RUN the program so far.

3.2 Moving the aircraft — animation loops

Right now our aircraft is sitting motionless in its place at the top left of our screen. We have to get it moving. Getting it moving comes even before we build in pilot controls. The craft must always be moving forward. A pilot doesn't have the option of stopping his plane!

We create the illusion of movement by changing the `PL`ace in which the aircraft stands. That is, we create a *loop* that returns to the `PL`ace where the aircraft is and updates that `PL`ace with the information about the new position. We'll put our update in line 214 and our return at line 400.

```
214 PL=PL+1
400 GOTO 214
```

Line 214 moves the aircraft one `PL`ace on and line 400 sends the program back to line 214 for same thing to happen again. The illusion is one of constant movement. Try a test `RUN`. You'll need your `RUN/STOP` key to stop it!

Well, the illusion isn't quite right yet. We've created movement but the craft is leaving a trail. The solution is to rub out the old craft before displaying the new one. Edit line 214 so now it reads:

```
214 POKE PL,32:PL=PL+1
```

As we know, 32 is the character code for a space. To rub out the old, we simply display a space in that position. It's like using whiteout. Try another test `RUN` using the new line 214. Let it run long enough for the aircraft to crash into the buildings before stopping it.

There are two important things to note:

- when our bomber flies off the right hand edge of the screen, it reappears on the left on the *next row down* and
- when it hits the buildings, nothing happens!

The question is then whether we want to keep these features in our program. In the case of the first one, I say yes. If the pilot doesn't pull his nose up every so often, he'll crash. In the case of the second, though, I think not! But we'll have to tell our computer what sort of action to take when such a collision occurs.

3.3 Collision detection

The first thing we've got to do is have our computer recognise that a collision has occurred. That is, it's got to check out each new position into which the plane flies to see whether it's occupied by a hazard of some sort — before displaying the plane in that position. A hazard, of course, is any PPlace that's not occupied by fresh air — ie character 32, the character for space. So we'll need a line that

- PEEKs into the PPlace next to be occupied by the aircraft
- checks it for a character which is not equal to (<>) character 32
- declares a crash if there's something other than space there.

```
220 IF PEEK(PL)<>32 THEN600
```

The last part of line 220 is reached only if there's a crash. It sends the computer off to line 600 of our program. So far there's no line 600, but there will be! At line 600 we'll start our routine for the crash. It will include sound and graphic effects, print CRASHED! on the screen and send the program to another subroutine that will let the user restart the game if he or she wants to.

Just for the present, you can type in these lines so you can test RUN the program so far. Later on we'll fill in the details so the effects will really happen!

```
600 REM CRASH
605 POKE PL,42:POKE PL+54272,0:REM
    EXPLODE SYMBOL
610 PRINT"BOOM NOISE"
615 PRINT"CRASHED!"
```

For the really high flyers, the words *DIVE BOMBER* will also be a hazard. If you want to, you can make this a different fate and call it something like LOST IN THE CLOUDS. The only hazard at any location less than 1500 is the title in the sky, so no real 'collision detection' is required. You just have to add a line:

```
602 IF PL<1500 THEN PRINT"***LOST IN
    THE CLOUDS":STOP
```

On the other hand, if you don't want the writing in the sky to be a hazard, you can use the same trick to discount its effect.

```
602 IF PL<1500 THEN 240
```

This sends you back to the flight routine without apparent interruption.

4

Bombs away!

4.1 Releasing the bomb

The nature of your craft is that it can only release one bomb at a time. It will not accept a command to drop a bomb until the previous bomb has fallen. So before the bomb release sequence can be started, a check must be made to see that there's not a bomb still in the air:

```
300 REM BOMB  
302 IF BO=0 THEN 350
```

The actual bomb release sequence now starts at line 350. The first thing we'll need is a key to serve as the bomb release button. The function key, f7 is a convenient one for right handers. Its code is key 3. If you're left handed, you could use the Z key. Its code is key 12. The release sequence looks like this:

```
350 REM CHECK CONTROLS  
356 KE=PEEK(197)  
360 IF KE=3 AND BO=0 THEN DR=1:BO=1
```

Line 356 sets up a variable name for the contents of memory location 197. 197 stores the name of the key being pressed right now on the keyboard, so KEy is an appropriate name for it.

Line 360 now calls up the new variable KEy and checks to see whether it's holding 3, the key for bomb release. IF it is AND the last bomb has fallen (ie BOmb = 0) THEN the DRop routine is switched to 1 and we register that BOmb is underway (= 1). We do nothing more now while the program goes through its next flight step until it once again reaches bomb check at line 302.

4.2 Animating the bomb

This time we find (in line 302) that BOMB is really go, so we'll have to write some bomb dropping stuff in between lines 302 and 350. The first of these lines will set the bomb position (BP) so that the bomb will remember which column to fall down while the aircraft itself continues to move across the screen to the right.

```
304 IF DR THEN BP=PL:DR=0:REM BP=BOMB POSN
```

Once we have our starting bomb position, we cancel the DROp indicator. We say DROp = 0.

Line 304 sets up a variable name for the bomb position (BP) and gives it the starting position PL — the PLace which the aircraft itself occupies at the time the DROp sequence starts. It does this, however, only IF the variable DROp is operative (ie it has a value greater than zero).

The computer will know whether DROp has been activated from the instructions in line 360. Remember that line 302 skipped the CHECK CONTROLS sequence at line 350 every time until the right key was pressed.

DROp therefore signifies the start of a bomb falling. Once the sequence has started, DROp must be reset to zero (DR=0) so that the bomb won't start again automatically in the next loop. Throughout its fall, BOMB retains the value of +1.

Variables as statements

In line 304 we used the variable name DR as a statement. That is, we used it without giving it a value. We just said IF DR THEN. The computer handles this in the same way as if we had said IF DR <> 0 THEN. In other words, the statement DR is counted as true if DR has a value of anything other than zero.

Now for the bomb itself. We'll use character 46, the full stop. And for the falling action, we lower it a row at a time (40 screen addresses on from its previous position) erasing it from its old position each time as it goes.

```
310 POKEBP,32
314 BP=BP+40
320 POKE BP,46:POKE BP+54272,0
```

Bombs away!

Line 310 POKes the old bomb position (BP) with a space (character 32).

Line 314 replaces the value held in BP (its screen address) with a value 40 addresses further on — ie the same column in the row immediately below.

Line 320 POKes the new BP with the character code for the full stop (character code 46) and colors it black.

Our next step must be to register when the bomb has reached the bottom row of the screen. This is necessary for two reasons:

- if we don't do something to stop it when it reaches the bottom row, it will fall into memory areas beyond 2023 and destroy our program! (Don't test RUN your program now unless you SAVE it first!)
- until we stop it, we can't release another bomb.

4.3 Stopping the bomb

One way of telling when the bomb has reached the bottom row would be to check the address of the bomb against each of the screen addresses for the positions along the bottom row. It's easier, though, if we fill the bottom row with characters of some sort and then handle it as we did the collision detection sequence for the aircraft with buildings.

So first of all, we'll fill the bottom row with solid dark grey blocks so it will actually look like ground.

```
117 FORF=1984 TO 2023:POKEF,160:
    POKEF+54272,11:NEXT
```

Line 117 sets up a FOR...NEXT loop to handle this repetitive task. It instructs the computer to POKE the character code for reverse space (160) into each of the screen addresses from 1984 to 2023 and to color them grey (11). F is just the name we've given to a variable (a memory cell) to hold the addresses to be POKed and to check them off as they are.

To check for contact with the ground during the bomb's fall, we use the collision detection routine. We PEEK the bomb's position (BP) and IF it equals 160 (the character we've used for ground) THEN we say the bomb has finished its fall — ie BO=0.

```
317 IFPEEK(BP)=160THENBO=0
```

Bombs away!

0

Once the bomb has completed its fall, of course, we want to be able to release another one. So:

```
310 IF B0=0 THEN 350
```

We've sent the program back to the start of the sequence for checking for a new bomb release. If you like, you can try a test RUN of the sequence now.

Note: a little later in the book we'll 'fold in' lines for the sound and graphic effects needed for the bomb's fall and its contact with the ground and buildings.

5

Pilot control

Because we can't release a new bomb until the previous one has completed its fall to the ground, it stands to reason that we could release more bombs if we could fly lower. The lower we fly, the less time it takes for a bomb to reach the ground.

And a little later on, we're going to vary the heights of the buildings in our cityscape. That will mean that we will be more effective if we can dive low for the low buildings but that we'll also need some way of climbing again to avoid crashing into the higher ones.

First of all, we'll need to choose control keys for up and down. We'll use the function keys f1 for up and f3 for down. f1 is key number 4 and f3 is key number 5.

Now we must instruct the computer what to do when it recognises either of these keys being pressed. We fold our new lines into the PLANE MOVEMENT sequence we commenced at line 200.

```
215 KE=PEEK(197):IF KE=4 THEN PL=PL-40
217 IF KE=5 THEN PL=PL+40
```

Line 215 sets up the variable name KEy for the contents of memory location 197 (the place that holds the name of the key being pressed). It then calls up the variable KEy to see whether it's holding code number 4. IF it is, THEN the value of PL (the aircraft's PLace) is reduced by 40 addresses. Reducing PL by 40 addresses takes it up a whole row.

Line 217 works in the same way except that this time we don't have to set up the variable name KEy for location 197. It's been done already. And this time, if the key pressed is code 5, the aircraft's PLace is increased by 40 addresses, taking it down a whole row.

Now test RUN the program so far. Test your elevation controls, f1 for up and f3 for down. Beware: don't attempt to fly your craft above the area displayed on the screen! If you do, you'll crash into the areas of memory held below 1024 and destroy your program.

You can prevent such an accident occurring with this line:

```
216 IF PL<1024 THEN PL=PL+40
```

In other words, IF the aircraft's PPlace is less than 1024 (the first address on the top row of the screen display) THEN 40 addresses will be added to it — it will be automatically relocated on the next row down. Test RUN again now to see whether you can fly above the screen.

Note: the control keys we've used for up and down elevation of the aircraft will be convenient for right handers. If you're left handed, substitute key Q for up and key A for down. Their code numbers are 62 and 10 respectively.

6

Landing

A real challenge for players of *Dive Bomber* is the landing sequence. The object of the game, of course, is to create a flat landing strip long enough to allow the aircraft to taxi to a halt without crashing into anything.

6.1 Touchdown

The first thing we've got to do is make the computer recognise the fact that the craft is coming in to land. The condition for the landing sequence will be that there's solid ground occupying the address *immediately below* the craft's PLace. Solid ground is character 160. We have to imagine that the craft's undercarriage will be lowered onto it!

```
250 IF PEEK(PL+40) <> 160 THEN 300
```

Line 250 is only a slight variation on the routine we used earlier for collision detection. The variation is that this time we're PEEKing not the craft's actual PLace, but the PLace 40 screen addresses on (PL + 40) to see whether it's occupied by a particular character.

There's something else we're doing here that's also worth noting. We're saying that IF the character in PL + 40 is *not* ground (160), THEN the computer should go immediately on to the start of the bomb release sequence (line 300). That's because we don't want to slow things down until we've really landed.

6.2 Taxiing

Having ‘touched down’ we must allow the craft to run to a stop. The question is, how many positions must it taxi through before it does stop? I’ve chosen 15. You can make it harder or easier, as you choose, by varying the number. Anyway, for 15 positions, we’ve got to set up a loop to count the 15 addresses on from the last PL+ 40 in line 250.

```
260 TA=TA+1:REM TAXI COUNT
262 IF TA<15 THEN300
270 PRINT"■■■■■■■■■■LANDED SAFELY!":GOTO 800
800 REM FINISH
```

Line 260 sets up the new variable TAXi and tells the computer to keep adding 1 to its value. Each time the loop is executed, TAXi goes up by one.

Line 262 is the control line. IF TAXi is less than 15 (meaning the loop has been executed fewer than 15 times) the program goes straight on to line 300 and continues the loop. This is useful in case we want to abort the landing for some reason. When TAXi becomes equal to 15 (indicating the craft has moved on 15 addresses) the program goes to line 270.

Line 270 signals that you’ve landed safely and sends the program to the routine that prints the score and allows the user to restart the game.

Before you test run

Before you test RUN the newly developed landing capability of your craft, you should get the buildings out of the way — just temporarily! So for your test RUN, change line 108 so it reads:

```
108 FOR RO = 24 TO 24
```

6.3 Slowing down the taxi

The landing sequence would look more convincing if the aircraft actually slowed down before it stopped. To do that, we need a delay loop between each of the craft’s 15 ground positions. And the delay between each position will have to be slightly longer than the one that went before it, if the illusion of slowing down is to be effective. In other words, our delay loop must contain a variable that increases as the landing proceeds. The obvious variable to use is TAXi. If the craft is on the ground, let’s count to twenty times TAXi between each position.

Landing

```
261 FOR F=1 TO 20*TA:NEXT
```

Line 261 is really just a 'time waster' for your computer so it won't move the plane quite so quickly. You've told your computer to set up a variable F and to add 1 to it ($20 \times \text{TAxi}$) times. The word NEXT sends the computer back to the FOR to carry out the operation for the next value of F (ie one more than the last time).

The reason that the delay is a little longer each time between the landing positions is that TAxI is greater every time. It's worth 1 the first time and 15 the last time.

6.4 A challenge

If you want to make the game more sophisticated, let the bomb create a crater when it misses a building and hits unoccupied ground. These craters will then be hazards for landing attempts. The craft will crash if it taxis over a crater (ie IF PEEK(PL+40)= crater character). If you decide to try this enhancement, SAVE your program first and remember to stop the bomb when it's made its crater.

Note: don't forget to replace line 108 with its original version:

```
108 FOR RO = 18 TO 24
```

7

Improving the graphics

7.1 The aircraft

As it is, our aircraft is too short to look enough like the real thing. The solution is to make it a double character. It's an easy matter to edit lines 240 and 242 to take in the second character:

```
240 POKE PL-1,252:POKE PL,252
242 POKE PL+54272-1,0:POKEPL+54272,0
```

The symbol for the aircraft will now look like this:



And because we've got a double character for the aircraft, we'll need a double clear space to erase it as it moves forward. Our line for printing space over the 'old' craft position was line 214. So now line 214 becomes:

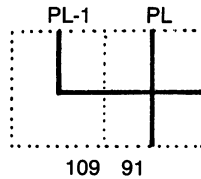
```
214 POKE PL-1,32:POKE PL,32:PL=PL+1
```

Test RUN the program now to check the effect. We certainly have a longer aircraft, but there's something 'off' about the erase and reprint routine. The craft appears to be 'worming' its way across the screen! Fortunately, there's something we can do to correct it. If we print the *right* side of the craft first, we'll create a *stretching* effect rather than the *concertina* effect we now have. All you have to do is edit lines 240 and 242 again so they look like this.

```
240 POKE PL,252:POKE PL-1,252
242 POKE PL+54272,0:POKEPL+54272-1,0
```

Now that we've sorted out the size of the craft and its movement, we can do something about the design of the craft itself. I like the combination of characters 109 and 91.

Improving the graphics



Line 240 now becomes:

```
240 POKE PL,91:POKE PL-1,109
```

7.2 The city skyline

The city would look more real if the buildings were of different heights. The way we do this is to apply the computer's random number generator to the rows holding the buildings.

```
106 BU=INT(RND(5)*7)+18
```

Line 106 sets up the variable name BUILDING and gives it the value of a set of numbers between 18 and 24, chosen at random. You may have to do a little 'upside down' thinking to visualise the effect: a number 24 will create a building of no height at all because 24 is the bottom row on the screen and a number 18 will create a skyscraper because 18 is the number of the top row of the building set.

Random numbers on the Commodore 64

As an example, consider how we make up line 106. Let's build it up in steps:

RND(5) will randomly give us any number less than 1, for example, .7362438.

RND(5)*7 will multiply the above by 7, giving us any number less than 7, for example, 5.326487.

INT(RND(5)*7) will give us any *whole number* less than 7, that is from 0 to 6.

INT(RND(5)*7)+18 will give us any whole number from 0+18 to 6+18, that is from 18 to 24.

(The 5 in RND(5) could be any positive number.)

In this way you can control the range of random numbers you get.

Now that we've done that, however, we'll need to change our existing line 108 so that it takes in the new variable **B**uilding.

```
108 FOR RO = BU TO 24
```

That means that the top 'storey' for any of our buildings will now be one of the randomly generated numbers from 18 to 24 (and the higher the number, the lower the building!)

7.3 A facelift for the buildings

Now we'll vary the look of the skyscrapers by allowing ourselves more than one sort of block from which to build them. The 'block' characters are usually the reverse of other symbols. I like the symbol for *reverse quotes* — character 34 (plus 128 for reversing).

Our old block was 250 so now we'll let the computer randomly choose which blocks to use for each building — 162 or 250.

```
107 BL=INT(RND(7)*2)*88+162
```

Line 107 sets up the new variable name **B**lock. The expression `INT(RND(7)*2)` will yield 0 or 1. The value of **B**lock can be either:

- 0 times 88 + 162 = 162 or
- 1 times 88 + 162 = 250

The 0 and the 1 in the sums are the numbers generated by the expression `INT(RND(7)*2)`. Remember that `RND(7)*2` won't ever return a 2, which is the outside *upper limit* of the series. The numbers 162 and 250, of course, are the numbers of the block characters that we want. If you haven't guessed how we arrive at the 88, here's how: $250 - 162 = 88$!

Now to complete the building routine, we must edit line 111 to include the new variable **B**lock.

```
111 POKE PL,BL
```

8

Scoring the game

8.1 The basic score

The first question we've got to concern ourselves with is the nature of the game itself. What is the challenge? How can one player beat another? What constitutes 'doing better'? Well, the fewer bombing runs you need before you can land safely, the better pilot/bombardier you are.

We'll start by allowing a maximum of 40 runs — that's one building per run and should be enough for even the slowest players. Now for scoring we can award a point for every run *not* needed. That is, a player's score will be 40 *minus* the number of runs he or she takes. We can't use the word RUNS itself as a variable name because RUN is a word reserved for the BASIC language. We'll use the word PASSES instead.

```
804 PRINT"#####YOUR SCORE="(40-PA)
```

When the program reaches it, line 804 will print on the screen the words YOUR SCORE= and then the result of the sum, 40 minus the value of the variable PASSES. We haven't set up the variable PASSES yet, so let's do that now.

A pass is 40 loops — ie a set of 40 screen addresses. That's how many addresses the aircraft travels through each time it flies across the screen. We'll count the loops themselves at line 218 and then convert LOOPS to PASSES at line 802.

```
218 LO=LO+1:REM LOOP COUNT
802 PA=INT(LO/40):REM 1 PASS = 40 LOOPS
```

Line 218 sets up the variable name LOOP and adds one to it every time the PLANE MOVEMENT sequence is executed.

Line 802 sets up the variable name PASSES and gives it the value of the number of LOOP divided by 40. The word INT tells the computer that we only want it to return us a whole number as the answer to the sum. No decimals thank you!

8.2 Landing skills bonus

We can fine tune the scoring by allowing a bonus for any buildings left standing after the pilot lands the aircraft. This is actually a 'landing skill' bonus. To find the buildings left standing, we simply make a subroutine to count all the non-empty positions on the first storey level. If there's something there, the rest of the building must also be intact!

```
840 FOR F=1944 TO 1983
842 IF PEEK(F) <> 32 THEN LE=LE+1
844 NEXT F:LE=LE-2:RETURN
```

Line 840 sets up the loop containing each of the screen addresses for the row containing the first storey level of the buildings. It's the first row above ground level — ie row 23.

Line 842 PEEKs into each of these addresses (now called by the variable name F) and IF it's anything other than a space (32), adds a score of 1 to a new variable it's set up by the name of LEft. LEft is the name we're giving to the buildings that are *left over* after the landing sequence.

Line 844 contains a number of important elements.

- NEXT sends the computer back to line 840 to ensure that all the values of F will be called in turn.
- LEft = LEft - 2 is needed because the aircraft itself fills two of the positions on the ground floor row. You can't count the characters for the aircraft as buildings left standing!
- RETURN sends the program back to the place *immediately following* the command that sent the computer to its scoring subroutine. The line that sends the computer on to a subroutine contains the command GOSUB. We haven't written that yet so we'd better do it now!

```
803 GOSUB840
```

And because we've altered the scoring with the bonus for buildings left standing, we'll have to edit line 804 to take in the extra information:

```
804 PRINT"#####YOUR SCORE="(40-PA)+LE
```

Scoring the game

8.3 Remembering the highest score

It's always useful if the computer can recall the previous best score achieved on the game. It solves disputes and gives a player something against which to compare his or her score.

The problem about this, though, is that every time we RUN a program from the start, we lose the values of all our variables. Restarting a program automatically clears the values stored under the variable names. Fortunately, there's a solution to the problem. The solution is to store the old High score in an address that is outside of the normal program storage area in the computer's memory. The address we'll use is 830 (it's the cassette file buffer). A number stored in address 830 will remain there as long as the computer is switched on — unless we also switch on our cassette drive.

Here then is how we can store the High score between RUNs.

```
805 IF(40-PA)+LE>HI THEN HI=(40-PA)+LE:
    POKE830,HI
```

Then we can display High score with:

```
800 REM FINISH
801 HI=PEEK(830):PRINT"#####HIGH SCORE="HI
```

Line 800 is simply the 'header' for the whole of this module to do with FINISHing the game. It's for people reading the lines of code we've written, not for the computer to do anything with.

Line 805 replaces the score held in 830 with the new score IF the new score is greater than the old score. It actually sets up the new variable name HI while it's carrying out the IF. . . THEN command.

Line 801 PEEKs into memory location 830 and PRINTs the number it contains following the words HIGH SCORE = . There are some cursor rights and cursor downs there to centre the words nicely on the screen.

8.4 A handy restart

A nice feature of any game program is a handy restart facility — something that will save the player having to type in the whole re-run command.

```
807 PRINT"#####PRESS M FOR MORE"
810 GET A$:IFA$<"M"THEN810
820 RUN
```

Line 807 simply displays the invitation **PRESS M FOR MORE**. There are the usual cursor rights and cursor downs and a command to print it in blue. It will be displayed immediately following the message **CRASHED!** so you'll also need to edit line 615.

```
615 PRINT "■■■■■■■■■■CRASHED!" :GOTO807
```

Line 810 uses the GET command to assign the next key press to the variable A\$. It then instructs the computer to move back to the start of line 810 IF the value of A\$ is not (ie <>) M. That means that if M is the value of A\$, the computer will pass on to line 820, the line that restarts the program. If it's not, the program will sit at line 810, displaying PRESS M FOR MORE until you stop the program with the RUN/STOP key.

In line 820, the command RUN automatically performs the important task of resetting to zero the variables for TAx_i, B₀mb, D₀rop and KEY.

9

Zaps and booms

The only things left to do now are the sound and graphic effects for the various explosions that occur throughout the game. We'll need a 'whine' for the bomb, a large boom and flash for the crashing of the aircraft and a smaller boom and flash for the impact of a bomb.

9.1 Variable names for sound and color

First of all, to save ourselves lots of typing later on, we'll set up some abbreviations for the sound and color addresses.

```
50 VO=54296:PI=54273
52 DE=54277:GA=54276
54 BA=53281:BR=53280
```

Line 50 sets up variable names for the addresses for VOlume and PIitch, two necessary elements of any sound we want to make occur.

Line 52 sets up variable names for the addresses for DEcay (the fading away of a sound) and GAte (to start and stop the sound). The GAte address also controls the waveform or type of sound we want.

Line 54 sets up variable names for the addresses for BACKground and BRder colors. Notice that we've left out the 0 from the regular spelling for *border*. That's because we've already used the variable name BOmb and as you'll recall, the computer only reads the first two characters of a variable name.

We use the line numbers at the very start of the program so that no matter where we later make use of the abbreviations, the computer will recognise them. Notice that now, if we wanted to, we could go back to line 90 and use the abbreviations for the full addresses appearing there for BACKground and BRder.

9.2 Bomb whine

The first step is to set up the actual whine of the bomb — ie the way the sound starts out high and gradually falls away as the bomb falls further away from the aircraft.

```
56 POKE VO,15:POKE DE,15
```

Line 56 sets the VOLUME and DECAY elements full on. Each of them has a range of 0-15 where zero is off and 15 is maximum on.

Next, we have to choose a PITCH for the sound and turn the GATE on with a setting for the particular waveform we want for the sound. We'll start out with a PITCH of 130 (that's pretty high) and a GATE value of 33. The number 33 is actually read by the computer as two separate numbers: a 32 for the waveform (it's a brassy sound) and a + 1 which is the signal to turn the GATE on.

```
362 IF DR THEN WH=130:POKEPI,WH:POKEGA,33
```

Line 362 fits into the program immediately after the keyboard scan for the bomb drop. It sets up the new variable WHine and IF a bomb drop has been detected, gives WHine the value of 130. It then goes on to POKE PITCH with the value of WHine and to POKE GATE with the value of 33. In summary, it starts the whine effect.

Our next step, then, will be to lower the PITCH of the WHine progressively as the bomb falls. To achieve this we lower the value of WHine with the lowering of the bomb through the air. We include this within the bomb movement sequence.

```
315 WH=WH-2:POKE PI,WH
```

Line 315 sets up a loop to decrease the value of WHine by 2 on each 'step' of its fall. It POKES the new, lower value, of WHine into the address called by the variable name PITCH.

Now when the bomb reaches the lowest point in its fall (the position identified in line 317) we must close the GATE to cut off the sound.

```
317 IFPEEK(BP)=160THENBO=0:POKEGA,32
```

We've expanded line 317 to close the GATE. Notice that to close the GATE we've POKED the number 32. That is, we've subtracted the 1 that opened it. The GATE is open when its value is an *odd* number and closed when its value is an *even* number.

Zaps and booms

Waveform settings for GAtE (address 54276)

Triangle	Sawtooth	Pulse	Noise
16	32	64	128

Adding 1 to any of these numbers opens the GAtE at the same time as it sets the desired waveform.

9.3 Boom! The aircraft explodes

We'll set up our BOOM sequence as a subroutine starting at line 700. The first thing we'll need to do, then, is include a GOSUB instruction at line 612, just before the instruction to print CRASHED!

```
612 GOSUB700
```

At this stage we should also delete our temporary line 610. We're going to provide a real boom effect.

```
700 REM BOOM
701 POKEDE,11:POKEPI,3:POKEGA,129
702 FORF=1TO8
703 POKEBR,0:POKEBA,2
705 FORQ=1TO50:NEXT
707 POKEBR,5:POKEBA,14
708 FORQ=1TO50:NEXT
709 NEXTF:POKEGA,128
710 RETURN
```

Line 701 creates the sound effect. First of all it POKes DEcay with 11. It's not the maximum 'dying away' effect you can get but it sounds about right for an explosion. Then it POKes Pltch with 3, which is almost as deep as you can go. GAtE is POKed with 129 (128 + 1), the waveform for *noise*. We don't want a musical sort of tone!

Line 702 sets up a loop to have everything in lines 703, 705, 707 and 708, repeated 8 times.

Line 703 POKes the BRder with 0 (black) and the BAcground with 2 (red).

Line 705 is a delay loop, there to slow down the color changes between lines 703 and 707.

Line 707 POKes the BRder with 5 (green) and the BAcground with 14 (light blue).

Line 708 is another delay loop, there to slow down the color changes between lines 707 and 703.

Line 709 closes the loop set up in line 702. It also contains the command POKE GAtE 128, to close the GAtE and shut down the sound.

Line 710 sends the program back to the line immediately following 612 — ie, line 615, which prints CRASHED! and sends the program off again to line 807, the PRESS M FOR MORE invitation.

9.4 Bomb hit: a shorter boom

We'll set up this shorter boom sequence as a subroutine starting at line 750. Again, the first thing we need to do is include a GOSUB instruction in the right place to send the program to it when it's needed. The right place is when our bomb position contains a building block (162 or 250). We have to go back to the bomb fall sequence to find it.

```
316 IF PEEK(BP)=162 OR PEEK(BP)=250 THEN GOSUB 750
```

Now we can include the BOMB HIT subroutine:

```
750 REM BOMB HIT
751 POKE DE, 11 : POKE PI, 4 : POKE GA, 129
753 POKE BR, 5 : POKE BA, 14
755 FOR Q=1 TO 5 : NEXT
757 POKE BR, 4 : POKE BA, 7
760 RETURN
```

Line 751 creates the sound effect. It POKEs DEcay with 11 again for the appropriate fade out of the boom sound. Then it POKEs PItch with 4, a slightly higher note than that for the aircraft's crash. GAtE, of course, is POKEd with 129, the code for noise and GAtE on.

Line 753 POKEs the BRder with 5 (green) and the BAcKground with 14 (light blue).

Line 755 is the delay loop to slow down the color changes between lines 753 and 757.

Line 757 changes the BRder to purple and the BAcKground to yellow.

Line 760 sends the program back to line 317, the line immediately following the one that sent the program off to this subroutine.

Now because we'll often need a run of these shorter booms and flashes in quick succession, we won't shut off the sound completely when the bomb first

Zaps and booms

strikes a building. Instead we'll close the GAte when the bomb finally hits ground level. That happens in line 317 of our program. So:

```
317 IFPEEK(BP)=160THENBO=0:POKEGA,32:POKEDE,15
```

We've added the close GAte instruction (32) and also restored our DEcay value to 15. That reminds us that we should also restore our screen and border colors to light blue and green respectively. It will be a simple matter to edit line 318 so now it reads:

```
318 IF BO=0 THEN POKE BA,14:POKE BR,5:GOTO 350
```


10

Joystick control

If you have a joystick and would rather use it than the function keys to control the game, you can change the control procedure from PEEKing the keyboard to PEEKing the movement of the joystick.

Plug your joystick into the *second port* on the right hand side of your computer (it's marked CONTROL PORT 2) and type the following line. It's not part of the program itself for *Dive Bomber* — we're doing it just to demonstrate something about the joystick.

```
950 JS=PEEK(56320):PRINT JS:GOTO 950
```

Line 950 sets up a variable name JS and gives it the value of the contents of address 56320, which holds the joystick 'POKEs' for its various positions. The rest of the line tells the computer to PRINT on the screen the values it finds for JS — and to keep doing it till we tell it to STOP.

So when you type RUN 950 you'll see a continuous stream of numbers on the screen. As you watch these numbers, move the joystick into its various positions and test the fire button. The joystick positions we're interested in are push (dive), pull (climb) and fire (drop bomb). For these positions we get the following numbers:

	FIRE	NO FIRE
UP	109	125
DOWN	110	126
LEVEL	111	127

Now we can fold these numbers straight into our program, substituting them for the values we have given our KEY variable.

```
356 KE=PEEK(56320)
360 IF KE<125 AND BO=0 THEN DR=1:BO=1
215 IF KE=109 OR KE=125 THEN PL=PL-40
217 IF KE=110 OR KE=126 THEN PL=PL+40
```

11

Dive Bomber The complete program

```
50 VO=54296:PI=54273
52 DE=54277:GA=54276
54 BA=53281:BR=53280
56 POKE VO,15:POKE DE,15
90 POKE53281,14:POKE53280,5
95 PRINT"J","轟轟轟轟 DIVE BOMBER *"
100 REM DISPLAY BUILDINGS
102 SC=1024:REM 1ST SCREEN ADDRESS
105 FOR CO = 0 TO 39
106 BU=INT(RND(5)*7)+18
107 BL=INT(RND(7)*2)*88+162
108 FOR RO = BU TO 24
109 PL=SC+RO*40+CO
111 POKE PL,BL
113 POKE PL+54272,9
115 NEXT:NEXT
117 FORF=1984 TO 2023:POKEF,160:POKEF+54272,
    11:NEXT
200 REM PLANE MOVEMENT
202 RO=5:CO=0
204 PL=SC+RO*40+CO
214 POKE PL-1,32:POKE PL,32:PL=PL+1
215 KE=PEEK(197):IF KE=4 THEN PL=PL-40
216 IF PL<1024 THEN PL=PL+40
217 IF KE=5 THEN PL=PL+40
218 LO=LO+1:REM LOOP COUNT
220 IF PEEK(PL)<>32 THEN600
240 POKE PL,91:POKE PL-1,109
242 POKE PL+54272,0:POKEPL+54272-1,0
250 IF PEEK(PL+40)<>160 THEN300
260 TA=TA+1:REM TAXI COUNT
```

```

261 FOR F=1 TO 20*TA:NEXT
262 IF TAC15 THEN300
270 PRINT"#####LANDED SAFELY!":GOTO 800
300 REM BOMB
302 IF BO=0 THEN 350
304 IF DR THEN BP=PL:DR=0:REM BP=BOMB POSN
310 POKEBP,32
314 BP=BP+40
315 WH=WH-2:POKE PI,WH
316 IF PEEK(BP)=162 OR PEEK(BP)=250 THENGOSUB750
317 IFPEEK(BP)=160THENBO=0:POKEGA,32:POKEDE,15
318 IF BO=0 THEN POKE BA,14:POKE BR,5:GOTO 350
320 POKE BP,46:POKE BP+54272,0
350 REM CHECK CONTROLS
356 KE=PEEK(197)
360 IF KE=3 AND BO=0 THEN DR=1:BO=1
362 IF DR THEN WH=130:POKEPI,WH:POKEGA,33
400 GOTO 214
600 REM CRASH
602 IF PLC1500 THEN 240
605 POKE PL,42:POKE PL+54272,0:REM EXPLODE SYMBOL
612 GOSUB700
615 PRINT"#####CRASHED!":GOTO807
700 REM BOOM
701 POKEDE,11:POKEPI,3:POKEGA,129
702 FORF=1TO8
703 POKEBR,0:POKEBA,2
705 FORQ=1TO50:NEXT
707 POKEBR,5:POKEBA,14
708 FORQ=1TO50:NEXT
709 NEXTF:POKEGA,128
710 RETURN
750 REM BOMB HIT
751 POKEDE,11:POKEPI,4:POKEGA,129
753 POKEBR,5:POKEBA,14
755 FORQ=1TO5:NEXT
757 POKEBR,4:POKEBA,7
760 RETURN
800 REM FINISH
801 HI=PEEK(830):PRINT"#####HIGH SCORE="HI
802 PA=INT(LO/40):REM 1 PASS = 40 LOOPS
803 GOSUB840

```

Dive Bomber

```
804 PRINT"YOUR SCORE="(40-PA)+LE
805 IF(40-PA)+LE>HI THEN HI=(40-PA)+LE:
    POKE830,HI
807 PRINT"PRESS M FOR MORE"
810 GET A$:IFA$<>"M"THEN810
820 RUN
840 FOR F=1944 TO 1983
842 IF PEEK(F)<>32 THEN LE=LE+1
844 NEXT:LE=LE-2:RETURN
```

12

How to write your own computer game

12.1 Variations on a theme

With a little imagination and only the programming skills you've learned in the creation of *Dive Bomber*, you can create a host of other entertaining games.

For example, if you color the sky black, make the buildings into unusually shaped alien dwellings and convert the aircraft to a spaceship, you'll have a *Planetary Raider*. And if you dot the sky with planets or meteors you'll have some hazards that will really test your flying skills!

In the same way, you can convert *Dive Bomber* to *Nautilus*, an atomic submarine, whose mission it is to invade the famous underwater city of Atlantis. Atlantis is no easy target because it surrounds itself with floating mines and every so often releases a random barrage of missiles that float upwards, destroying anything in their path.

You can make the games you create as hard or as easy as you like, depending on the skills of the players. You can increase or decrease the speed at which the craft moves. You can add a variety of different hazards and features that will score bonus points. One that I like very much is the ammunition dump. A hit on the ammunition dump takes out the buildings on either side of it. But when I include this feature in a game, I also build in a bomb blast hazard, just to keep things interesting. It's a trap for low flyers. A hit on the ammunition dump will also destroy the craft itself if it's close enough when the explosion occurs!

It's all a matter of imagination. Imagination to change the *interpretation* of the symbols and their movements and imagination to create the sound and graphic effects that make these games so exciting. You'll be surprised at how little of the basic program you'll have to change to create something really different.

12.2 Creating a new game

If you're creating a new game you'll find it a lot easier if you plan it thoroughly first. The more time you spend in planning, the smoother will flow your programming. Here's a good sequence to follow.

Write the story

Write (in English) the story line of your game. Keep it really simple to start with. You can add the enhancements later on.

Break it into modules

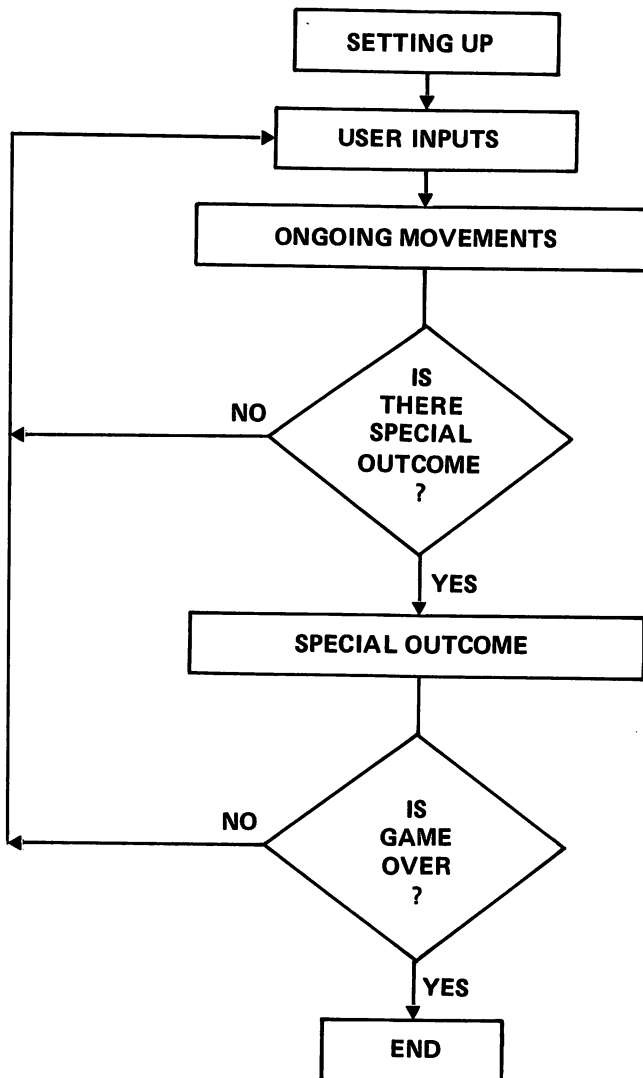
- Identify the characters in your story — ie the objects that will perform some action. List them and note what you want each of them to do.
- Describe the background against which the action will take place. Break it up into its various 'fields' — eg ground and sky — and note the items you're going to put into these fields. A rough sketch is a useful way to do it.
- Decide how the game will be scored — what's really the *object* of the game?
- Note how you want the game to end.

Plan the action sequence

The most important (and most difficult) part of a game program like *Dive Bomber* is the *action sequence*. It's the part where you usually have two or more objects moving in quick succession in response to controls, target hits or collisions.

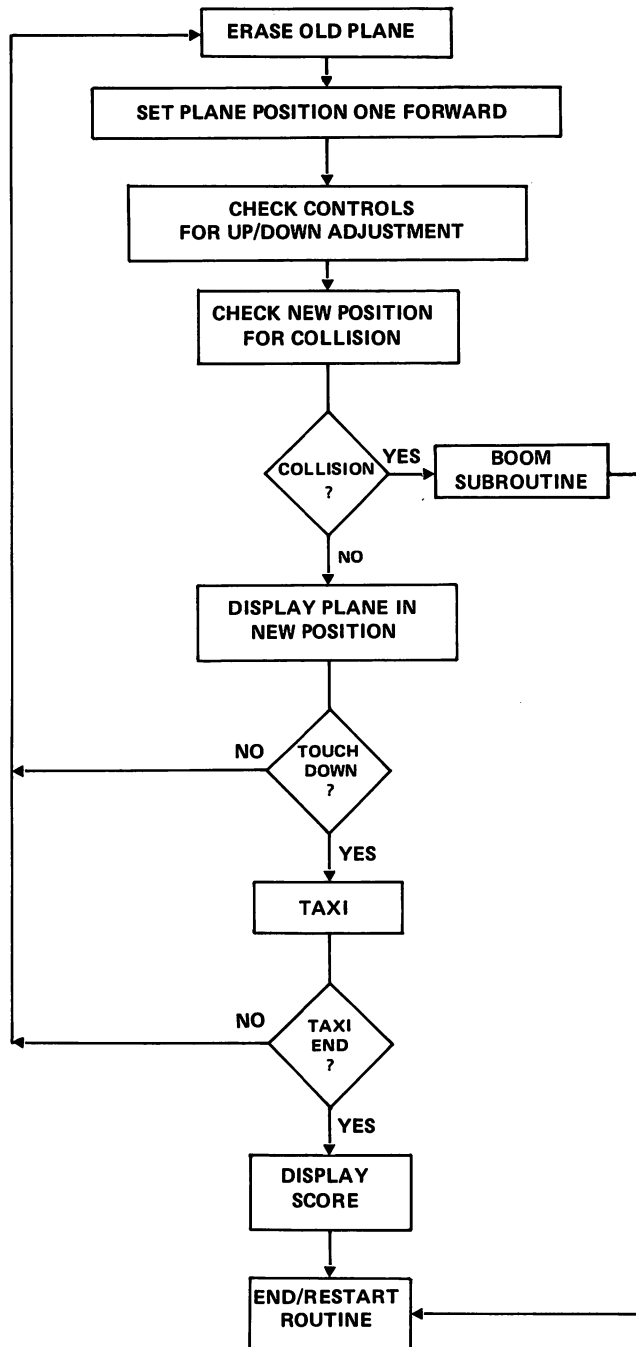
It's also the part you should work out first. There's nothing worse than having your background ready, your instructions written, your crash sequences working and your score displays ready to go — with an action sequence that won't go right. You can feel very silly indeed!

The action sequence in any game consists of the ongoing movements, the user inputs and the various outcomes. Diagrammatically, the sequence looks like this:



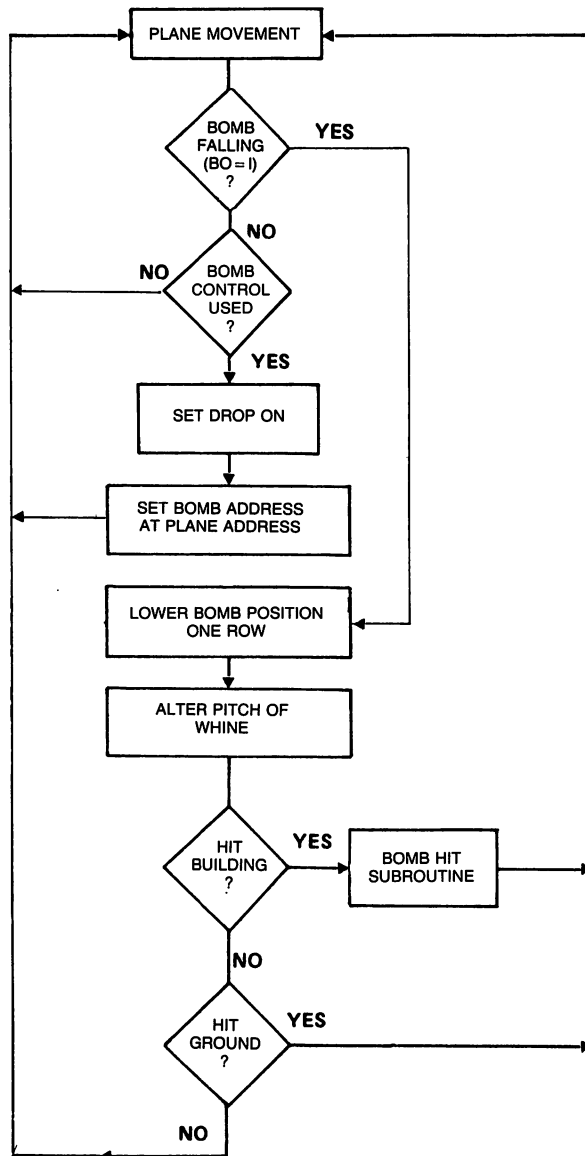
In *Dive Bomber*, the ongoing movements are the plane in its flight path across the sky and the bomb in its fall to earth. The inputs consist of the signals for climbing and diving and the signal for bomb release. The signals may come either from the keyboard or a joystick. The outcomes are collision, bomb strike and successful landing.

First of all we'll deal with the ongoing movements and in particular with the sequence for the control of the plane. Control of the plane requires these steps:

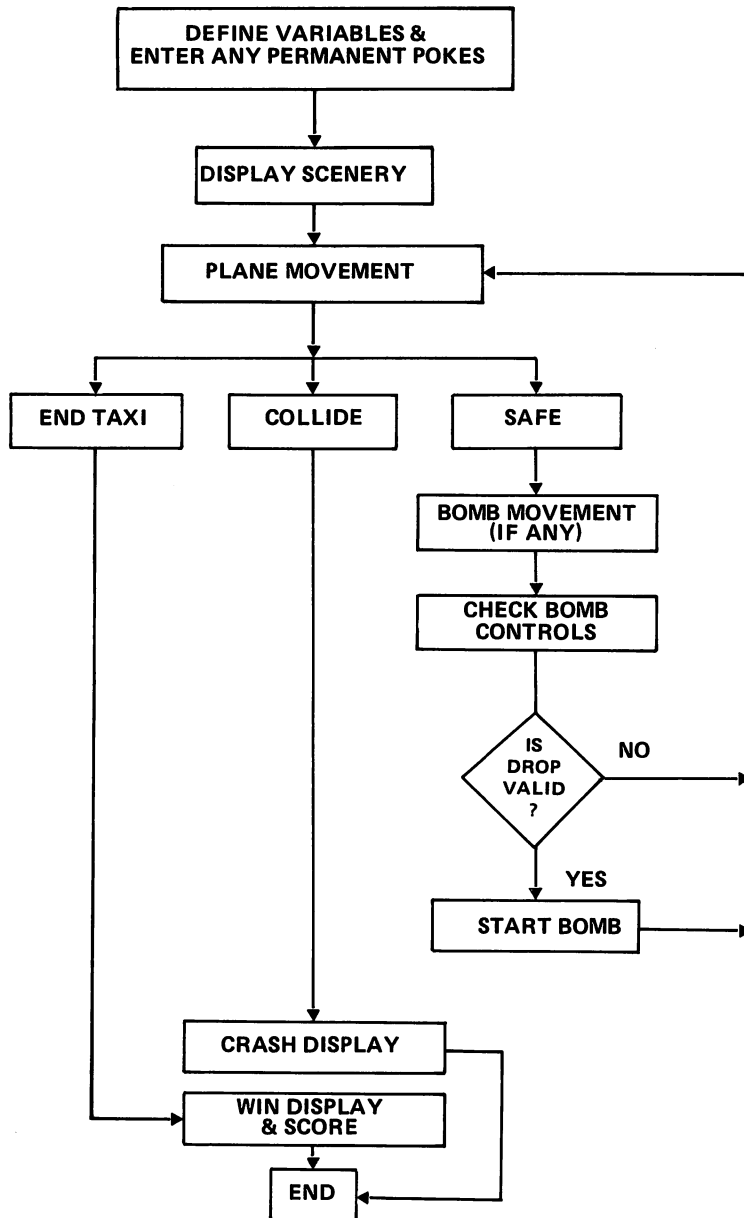


How to write your own computer game.

The bomb sequence is a little more tricky because we have to sort out whether we're starting a new bomb or simply lowering an existing bomb:



Once these two sequences — the plane control and the bomb control — are working, we can build the rest of the program around them.



This diagram will help you to draw a plan for the game your own imagination has created.

Write the code

Write your lines of code, *module by module*.

- **SAVE** your program (to tape or disk) as you complete each module and before you **RUN** it.
- **RUN** each module separately, as you complete it, to make sure that it works. Then **RUN** it in combination with any other modules you've already written. You may have to insert the word **STOP** at the end of a module to do this — so don't forget to remove the **STOPs** later!
- When you add a new variable, make sure its name is different from the names you've given to any other variables you've already used. Two variables are counted as being the same if their first two characters are the same. Test **RUN** to make sure.
- Avoid calling variables by names that include **BASIC** keywords — eg **RUNWAY**. Some others you should avoid are **TI**, **TI\$** and **ST**. They are reserved and have special meanings.
- Take special care with screen **POKEs**. Make sure that things can't run off the screen before you test **RUN**. To be on the safe side, **SAVE** everything you've written before you test **RUN**. If you do run off the screen, switch off the computer and reload your latest **SAVE** because important addresses are sure to have been fouled up by the run off.

Enhance the program

Now's the time to go back over the program, module by module, to insert the other things you've thought of to 'jazz up' your program — better graphics, extra hazards, player instructions and so on. As above, **SAVE** the program with its extra lines first and then **RUN** the whole program.

Dive Bomber places you at the controls of a super jet. Your mission is to destroy the city that lies below you. This book focuses on the development of this game and explains in detail every step needed to create its graphics, sound and action. However, while setting up and playing *Dive Bomber* you'll learn how to program your Commodore 64 — at the end of this game you'll be able to write your own games, even if you are a complete beginner!

ISBN 0 85896 159 8

Pitman